

AMES GRANT

IN-61-CR

**COOPERATING KNOWLEDGE-BASED  
SYSTEMS**

154095  
123

**Edward A. Feigenbaum  
Bruce G. Buchanan  
Principal Investigators**

**FINAL REPORT**

**Submitted to  
NASA-Ames Research Center**

**Contract NCC 2-220**

**August 10, 1988**

(NASA-CR-183109) COOPERATING  
KNOWLEDGE-BASED SYSTEMS Final Report  
(Stanford Univ.) 22 p

CSCI 09B

N88-29327

Unclass

G3/61 0154095

**Knowledge Systems Laboratory  
Department of Computer Science  
Stanford University**

## Table of Contents

1 Introduction	1
2 Concurrent Architectures	1
2.1 SIMPLE/CARE Multiprocessor Simulation System	1
2.1.1 The Design of SIMPLE/CARE	2
2.1.2 Architecture Design-time Interaction and Simulator Run-time Operation	2
2.2 LAMINA Programming Interface	4
2.2.1 Futures and Streams	5
2.2.2 LAMINA'S Models of Concurrent Computation	6
2.3 Polygon Problem Solving Framework	7
2.3.1 How Polygon matches the problem domain	8
2.3.2 How Polygon matches its target hardware	8
2.3.3 What we have learned to date	9
2.4 CAGE Problem Solving Framework	9
2.4.1 CAGE Design	10
2.4.2 Building applications in CAGE	10
2.4.3 Specifying Concurrency	11
2.4.4 CAGE Machine Model	13
2.5 CAGE, Polygon and LAMINA Comparative Experiments	13
2.5.1 The Experiments	13
2.5.2 Experiment Status	14
2.6 The AIRTRAC Application	14
2.6.1 Overall Application System Structure	14
2.6.2 AIRTRAC Organization	15
2.6.3 AIRTRAC Status	15
2.7 Multiprocessor Load Balancing Studies	15
2.7.1 Load Balancing Studies Status	16
3 Spatial Reasoning	16
4 Reasoning Under Uncertainty	16
5 List of Publications	17
5.1 Concurrent Computer Architectures	17
5.2 Spatial Reasoning	18
5.3 Uncertain Reasoning	19

## List of Figures

<b>Figure 1:</b>	<b>Graphic Structure Specification</b>	<b>3</b>
<b>Figure 2:</b>	<b>Design Time Interactions and Run Time Representations</b>	<b>4</b>
<b>Figure 3:</b>	<b>Overseer Instrument</b>	<b>5</b>

## 1 Introduction

This final report covers work performed under Contract NCC 2-220 between NASA Ames Research Center and the Knowledge Systems Laboratory, Stanford University. The period of research was from March 1, 1987 to February 29, 1988. Topics covered were:

1. concurrent architectures for knowledge-based systems;
2. methods for the solution of geometric constraint satisfaction problems;
3. reasoning under uncertainty.

The research in concurrent architectures was co-funded by the Defense Advanced Research Projects Agency (DARPA), as part of that agency's Strategic Computing Program. The research has been in progress since 1985, under DARPA and NASA sponsorship. This is the primary task under this contract, and the extensive treatment given below, relative to the other two tasks, reflects that fact.

The research in geometric constraint satisfaction has been done in the context of a particular application, that of determining the 3D structure of complex protein molecules, using the constraints inferred from NMR measurements. This work has also been in progress for several years, and has been co-funded by DARPA.

The research on reasoning with uncertainty has been largely carried out by Mr. David Heckerman, a Ph.D. candidate in our laboratory, under the supervision of the Principal Investigators and Dr. Edward Shortliffe.

## 2 Concurrent Architectures

This research is addressing the following questions:

1. Can multiprocessor computers be used to achieve significant execution speedup (two to three orders of magnitude) over serial machines for knowledge-based system applications?
2. What are the limiting factors in achieving speedup for such systems?
3. What are appropriate software models and methodologies for programming such systems?
4. What are appropriate hardware architectures for supporting such systems?

In the following subsections, we present the major components of our project and the current status of each.

### 2.1 SIMPLE/CARE Multiprocessor Simulation System

Simulation of systems at an architectural level can offer an effective way to study critical design choices if (1) the performance of the simulator is adequate to examine designs executing significant code bodies -- not just toy problems or small application fragments, (2) the details of the simulation include the critical details of the design, (3) the view of the design presented by the simulator instrumentation leads to useful insights on potential problems with the design, and (4) there is enough flexibility in the simulation system so that the asking of unplanned questions is not suppressed by the weight of the mechanics involved in making changes either in the design or its measurement.

SIMPLE/CARE (KSL 86-36) is a simulation system which satisfies these requirements. It forms the foundation for our empirical investigations of software architectures and hardware system architectures for concurrent knowledge-based systems. SIMPLE is a CAD (Computer Aided Design) system for hierarchical, multiple level specification of computer architectures

and includes an associated mixed-mode, event-based simulator. CARE is a parameterized, multiprocessor array emulation specified in SIMPLE's specification languages and running on SIMPLE's simulator. Our simulation system is in use by several research groups at Stanford, and it has been ported to several external sites including NASA Ames Research Center.

### 2.1.1 The Design of SIMPLE/CARE

The overall research problem motivating the development of both SIMPLE and CARE is the performance study of 100 to 1000-processor multiprocessor systems executing knowledge-based signal interpretation applications.

A set of constraints pertinent to this problem governed the design of SIMPLE/CARE. The applications represent significant bodies of code and so simulation run times are an important consideration. Moreover, the issues involved with the interactions of multiprocessor system elements are sufficiently unexplored prior to simulation that simplifications in the architectural model, specifically with respect to processor interactions, are suspect. This need for detail is, of course, in tension with the need for simulation performance. The ways that simulated system components are composed into complete systems is difficult to bound. Further, it is clear that the models of these components are elaborated over time and undergo substantial change as design concepts evolved. It is also clear that the ways of examining the operation of these components would change independently (and at a great rate) as early experience indicates what alternative aspect of system operation should have been monitored in any given completed run.

The design goals that emerged are (1) that the simulation system should support the management of substantial flexibility with regard to simulated system structure, function, and instrumentation and (2) that, in order to accomplish runs in acceptable elapsed times, the detail of simulation should be particularly focused on the communications, process scheduling, and context switching support facilities of the simulated system -- that is, on just those aspects of system execution critical to multiprocessor (as opposed to uniprocessor) operation.

### 2.1.2 Architecture Design-time Interaction and Simulator Run-time Operation

Encapsulation of the state of design components with the procedures that manipulate that state is one clear way to manage architectural design evolution. Such encapsulation partitions the design along well defined boundaries. Components (by and large) interact with other components only through defined ports. Connections between components terminate at such ports. When a system simulation is initialized, connections are traced so that for every port, the simulator knows the connected (terminating) ports together with their containing components. Once such initialization is complete, that is, throughout the simulation run, assertions about the state of a port of one component can be directly translated to assertions about the state of connected ports of other components.

Partitioning issues of system structure, component behavior, and instrumentation into separate domains of consideration helps in managing a design that is both fluid and complex. System structure, that is, the relationship between components, can be specified through use of an interactive, graphics structure editor and is largely independent of component function per se. Figure 1 shows an example of SIMPLE's structural editor.

Component behavior is encapsulated in a set of definitions pertinent to the given class of component. Each component in a SIMPLE specified simulated system is a member of a class defined for that component type. Instrumentation is automatically and invisibly made part of the definition of each simulated component that is to be monitored during a run. This is done by arranging that the class of every component to be monitored is a specialization of the general instrumented-box class. The basic data structures and procedures for monitoring simulated components and maintaining the organizational relationships between each component and its related instrumentation are inherited through this general, ancestral class and are thus made a separate, substantially independent consideration in the design.

A further partitioning of concerns is employed to separate out the definition of the application programming language interface and its support (as provided by CARE) from the underlying

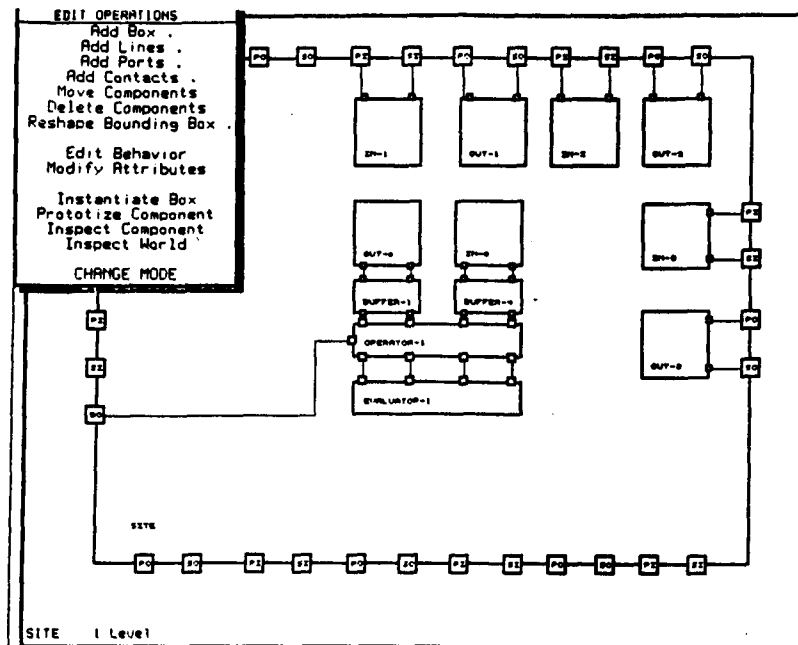


Figure 1: Graphic Structure Specification

information flow control governing component behavior. The behavioral descriptions of components (which are expressed as sets of condition/action rules) deal generically with gating information, independently of the structure of the information, between ports of the component and its internal state variables. This is separated in the component model definitions from the functions performed to create and manipulate the information so gated. The simulated implementation of the application programming language support facilities, on the other hand, relies only on the specifics of the information and its structure and plays no part in gating it between the components of the system. Changing the definition of the application language is thus done independently of changing component flow control behavior. The application programmer and the implementer of the application language interface may use whatever data structures seem suitable to them, be they numbers and keywords or procedure bodies and execution environments. The simulation system doesn't care.

The component probe definitions, that is, the specifications of what information should be captured for each component type, are separated from the descriptions of the behavior of such components. In designing for flexibility in the instrumentation system, it turns out to be important to further divide the information presentation from the information collection issues. The mapping from particular component probes to particular instrument panels and the transformations to be applied to the information as it passed from a given kind of probe to a given panel (and between panels) is captured in the instrument specification. This is a definition of what kinds of panels are included in an instrument, how they fit on an instrument screen, how they are labeled and scaled, and what information from which kinds of probes are displayed on each panel. The instrument specification also indicates what kinds of probes are to be connected to which kinds (that is, which classes) of components in the system.

Putting together all the definitions of components, component probes, panels, instruments, applications interfaces, and inter-component relationships is done in a set of design time interactions by a system architect. These interactions are used by the simulation system to generate efficient run time representations so that simulation performance goals can be met. Figure 2 illustrates the partition between design time interactions and simulation run time

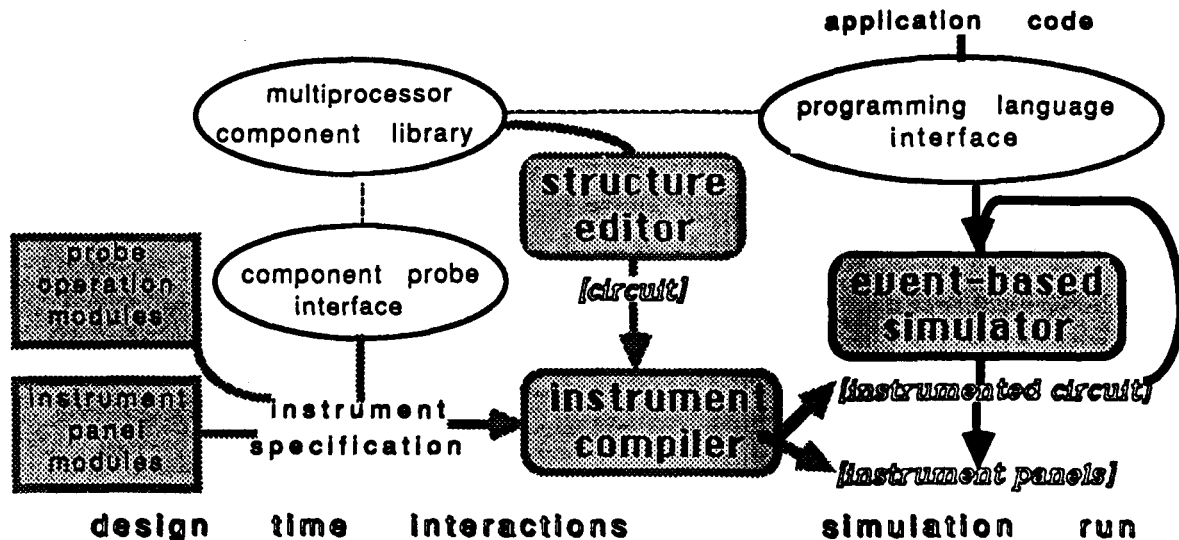


Figure 2: Design Time Interactions and Run Time Representations

operation. Structure editing pulls together components from the component library to produce a circuit. Associated with some components in the library, there are definitions for the syntax and underlying mechanisms of a multiprocessor applications language. These specify the interface used to provide the program input to the multiprocessor system being simulated. The definitions used to generate component probes are associated with each library component to be monitored. There may be several such definitions, each appropriate to measuring a different aspect of the associated component's operation. An instrument specification selects from these definitions, elaborates them with selections from a set of probe operation modules to include any pre-processing (for example, a moving average) to be calculated by the probe, and indicates under what conditions what information from the probe is to be sent to which panels of the instrument and how it is to be transformed and displayed there. Instrument specifications also partition the screen among the panels of the instrument. The end product of these design time interactions is an instrumented circuit and an instrument. The instrument comprises a set of instrument panels and a set of constraints relating them to the instrument screen. The instrumented circuit ties together instances of components, probes, and panels for a simulation run. Figure 3 gives an example set of instrument panels for a run.

For each defined class of component and its associated probes, the design time interactions produce code bodies that accomplish simulation operations during a run. It is an attribute of the underlying Lisp base of the simulation system that changes in these definitions have immediate effect even during a simulation run -- an important capability during debugging.

## 2.2 LAMINA Programming Interface

LAMINA (KSL 86-67) provides extensions to Lisp for studying expressed concurrency in functional programming, object oriented, and shared variable models of concurrent computation. The implementation of the support for all three computational models is based on the common notion of a stream, a datatype which can be used to express pipelined operations by representing the promise of a (potentially infinite) sequence of values. LAMINA also provides system support for the management of software pipelines and dynamic structure creation, relocation, and reclamation in a multiprocessor, multi-address-space system.

Algorithms and applications written in LAMINA may be run on the SIMPLE/CARE simulation system in order to study their execution on alternative multiprocessor architectures.

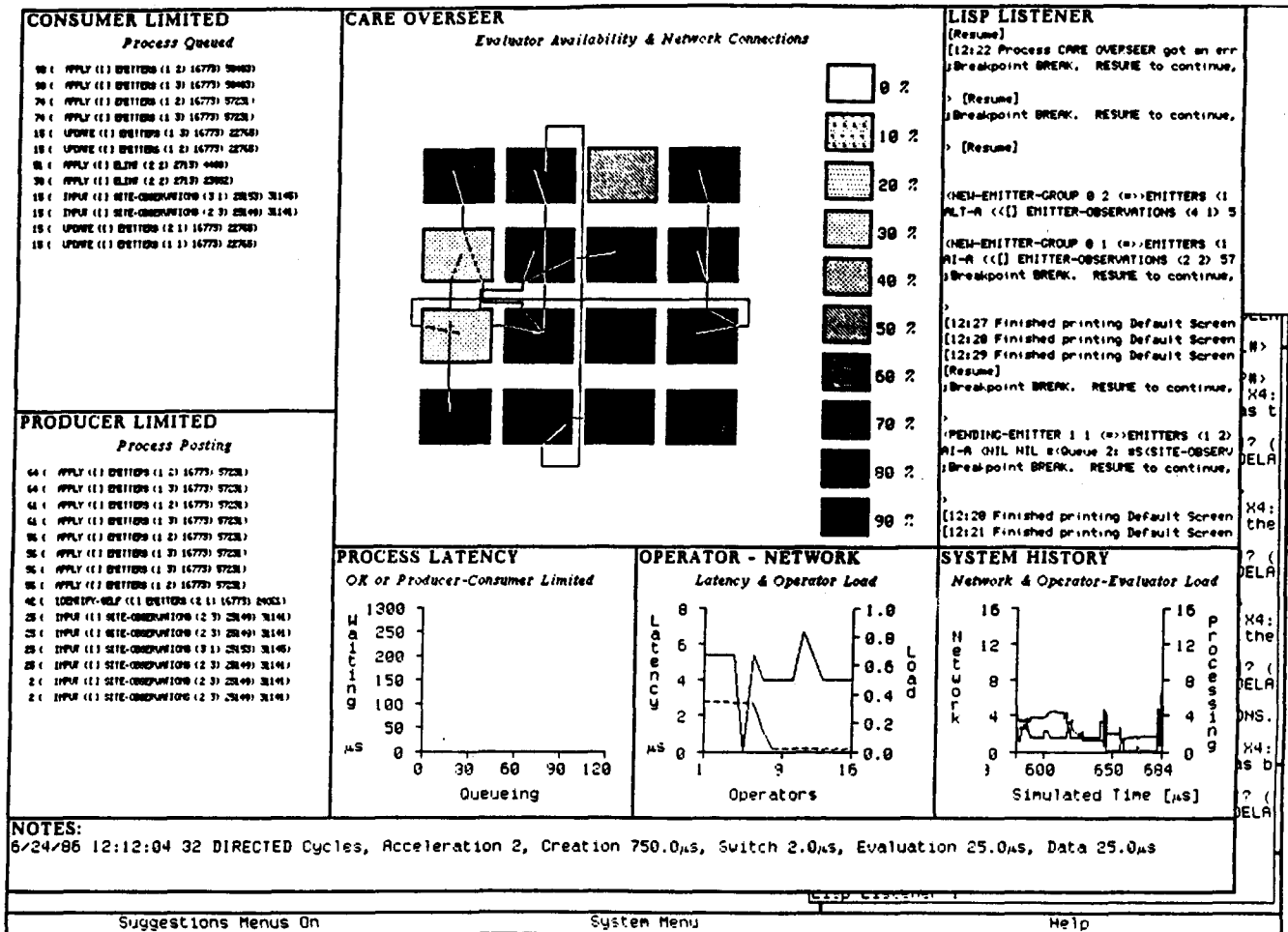


Figure 3: Overseer Instrument

### 2.2.1 Futures and Streams

Futures and streams provide the common ground between functional, object oriented and shared variable programming in LAMINA. They are fundamental to the LAMINA functional and object oriented programming regimes for parallel programming and, since they are the only mutable items passed as references (rather than structure values) between potentially concurrent computations in LAMINA, they are also used to build the mechanisms for shared variable computation.

Futures and streams represent promises for values. In LAMINA, futures can be used as placeholders in a computation while the values themselves are being eagerly produced by concurrent evaluations for consumption as available. Extending this idea, LAMINA defines a stream as an abstract data type which is a placeholder representing a sequence of eagerly produced but potentially unavailable values.

Some operators do not require the actual values promised by a stream or future in order to perform their work. For example, a constructor may create data structures that include streams as structure elements. The creation can be accomplished without accessing any of the promised values that the streams represent; referencing streams as placeholders is sufficient. Further,

streams, as sequences of potentially unavailable but eagerly produced values, can be used in LAMINA to build pipelines of computation connecting the producers and consumers of such values.

Streams may be arguments to or the results of function application. In LAMINA, streams are a primitive data type developed for use in an object oriented programming style and futures are a specialization of streams that represent only a single (potentially unavailable) value as required for the functional programming style. Streams and futures are always passed as references.

### 2.2.2 LAMINA'S Models of Concurrent Computation

Perhaps the style of computation most readily treated as concurrent is that of functional programming. LAMINA supports concurrent programming using this style by providing means (1) to spawn computations that will provide values to futures and (2) to accept such values in a computation -- scheduling the computation when they are available. The constructs defining the LAMINA interface for functional programming are:

- (future form) spawns execution of a lexical closure, that is, a procedure body to execute a given form together with an environment (determined by the rules of lexical scoping) in which to do the execution. This closure is executed (eagerly) on a randomly selected site. A future which will contain the value of the computation when it is available is immediately returned.
- (with-values future-bindings forms) spawns an evaluation on the local site to execute the closure corresponding to the forms. The evaluation is done within an environment that includes bindings for given variables to the values available for the indicated futures. The evaluation is deferred until all of the indicated futures have values that are not themselves futures. The immediate result of executing a with-values form is a future whose value will be supplied by the deferred evaluation.

In LAMINA's object oriented programming interface, an object encapsulates related state variables and is referenced throughout an application by that object's Self-Stream, a stream which is one of the object's state variables. Objects are allocated in a processor's local address space. To perform operations on an object, potentially involving and modifying its state variables, a task request posting consisting of a task selector and associated parametric values for the operation is sent to the object, that is, provided as one of the values of the self-stream for that object. Each of the task request postings that provide the values for the self-stream of an object is taken in turn from that stream and serviced by that object.

Task request postings are serviced atomically in the context of an object. Executions specified by such request postings are done without visible partition with respect to other operations on that object, that is, operations on any given object will not be interleaved. Each operation is thus defined to be independently atomic.

All the operations on an object done as specified by the requests are taken in turn from the object's self-stream. Each operation runs to completion. If an operation on an object is preempted (due, for example, to page faulting, schedule quanta lapse, or error condition), no other operation on that object will be started before the preempted operation is completed. However, operations on other objects may proceed normally. A stack is maintained for each preempted operation.

Shared variables are dealt with in LAMINA by treating them as references whose associated value may be mutated. A shared variable reference is constructed, accessed, and mutated by provided interface operations. Support for shared data pairs and arrays is also provided. For all these operations, execution is deferred and no other executions are performed by the initiating processor until the indicated operation is accomplished.

Shared queues (which are streams) are also provided. These queues are maintained in a processor's local memory. When a process reads from a shared queue, it is halted and

descheduled; execution is resumed when the requested data arrives. A simple spin lock is provided for busy-wait synchronization in the LAMINA shared variable interface.

Several utility operations are provided by LAMINA to specify computation (and storage) sites, dismiss computations, and provide a timeout facility for applications desiring one. LAMINA also provides simulation control facilities to initiate a CARE simulation, read the current simulation time, and do a computation without increasing the simulation time.

### 2.3 Poligon Problem Solving Framework

Poligon (KSL 86-19, KSL 88-04] is a framework for the development of Blackboard-like applications on a (simulated) multiprocessor. It consists of:

1. A compiler, which compiles a high-level description of the Blackboard's structure and the Knowledge to be applied by the system, to run on a distributed memory multiprocessor.
2. A run-time system which provides a debugging and testing environment for Poligon programs as well as run-time support.

Both the compiler and the run-time system are thoroughly integrated with the program development environment of TI Lisp machines, the machine on which the execution of Poligon programs are simulated.

Serial Blackboard Systems are implemented with the Nodes being represented as records on the Blackboard. The Knowledge is encoded in Knowledge Sources. These are typically compiled into procedures which are invoked by the Blackboard System's kernel. There is some form of scheduler for the Knowledge, which invokes one Knowledge Source after another. The Blackboard and the Knowledge Base both share the same address space, though they are functionally distinct. Knowledge Sources are "invoked" (executed) as a result of changes in the Blackboard placing that change event in a queue used by the scheduler. The scheduler repeatedly picks a Knowledge Source which is interested in the type of event at the end of the queue.

The design of Poligon has been motivated by the idea of trying to eliminate the bottlenecks that would be experienced if an existing, serial Blackboard System were to be parallelized only by the inclusion of "do this bit in parallel" constructs. The major changes from the serial blackboard model are listed below.

- The scheduling queue of a serial system is eliminated altogether in Poligon. This means that concurrent attempts to invoke Rules are not held up waiting for access to this shared data structure.
- Having a Knowledge Base, which is logically distinct from the Blackboard, is no longer necessary since there is now nothing to get between them to control the application of the knowledge. This allows all Knowledge to be attached to those Nodes that are interested in the Knowledge by the compiler.

These changes eliminate at one stroke the bottlenecks of the shared scheduler and the Knowledge Base to Blackboard interface. These changes allowed the development of the idea of the "Node as a processor" metaphor for parallel Blackboard systems.

Having eliminated the scheduling mechanism, however, one needs some means of determining when a certain piece of Knowledge should be invoked. It would be hopelessly inefficient to have all of the Knowledge executed all of the time, since most of the time it would find itself inapplicable. It was decided that a simple daemon-driven approach would be used to avoid this problem. This results in the Knowledge being directly sensitive to changes in the Blackboard and able to act immediately upon any such changes.

Existing Blackboard Systems often express the Knowledge in their Knowledge Sources as collections of Pattern/Action Rules. These are normally executed serially, in the lexical order

in which they are defined. Polygon on the other hand compiles Knowledge Sources away all together, allowing their constituent Rules to be executed in parallel.

The "Node as a processor" metaphor is itself a major step away from the normal means of implementing Blackboard Systems. This, however, is not enough. This would give us data parallelism, resulting from the large number of Nodes in the system being able simultaneously to execute Rules, whilst still failing to exploit the potential Knowledge parallelism. This is because each processing element is a uniprocessor capable of executing at most one Rule at a time. Polygon, therefore, goes beyond this simple model to one which would more accurately be called the "Rule invocation as a process" model. This allows the Polygon system to distribute concurrent Rule invocations to different processing elements.

The elimination of serializing components in a Blackboard system also eliminates those mechanisms which are normally used to preserve coherency in the solution. Clearly there is a trade-off which can be made between the amount of control and coherency preserving mechanisms and the amount of exploitable parallelism. Polygon is an experiment to explore one extreme of this spectrum. It remains to be seen whether the trade-off made in Polygon results in an overall improvement in system performance.

### 2.3.1 How Polygon matches the problem domain

Polygon is not a general purpose programming language, other than in the Turing Complete sense. It is specialized to support one computational model and that computational model, itself, has limitations on its sphere of reasonable applicability. It has been designed with applications such as real-time signal understanding and data fusion in mind, though applications outside this domain are being investigated.

The structure of the problem domain is one that requires the representation of a large number of distinct entities in the solution space. For example the vocabulary of the Elint problem domain (see Section ) is full of such things as aircraft, radar emitting platforms and radar track segments. Polygon provides a rich representation language in which these objects and specializations of them can be expressed. This allows the system to take full advantage of the mutual independence of any of the objects in the solution space to exploit parallelism.

### 2.3.2 How Polygon matches its target hardware

Polygon could, of course, run on any machine in principle. In practice, however, it has been designed with a CARE type of machine model in mind and has been optimized to take advantage of it. The grain size of the executable chunks in Polygon programs is designed to suit this model, i.e. each chunk represents, ideally, a few function calls. This makes it coarser grained than those systems that want to execute everything that can be in parallel, for instance data flow machines, but it is a lot finer grained than most other concurrent Blackboard Systems in which each processing element contains a complete Blackboard System.

The target machine model, being of the distributed-memory, message-passing variety including essentially no capability to pass references, strongly discourages shared variables or mutable global data of any sort and encourages a message-passing style of programming. The Polygon language is one in which the programmer is given an abstract view of programming using the Blackboard Problem-Solving model. The Polygon language has no construct for message sending at all, nor has it any primitives by which the user has access to the underlying architecture or topology. It is assumed to be the duty of the Polygon system or the target machine's operating system to look after such concerns. The Polygon compiler compiles its programs into the message passing primitives of the underlying system. This allows the efficient use of the underlying architecture, whilst still leaving the source program uncluttered by concrete details of the target architecture.

Polygon allows only global constants (but not variables) since these can be distributed at program load-time.

### 2.3.3 What we have learned to date

Experiments with Poligon are by no means complete, but we have learned quite a bit so far. Some of these lessons are enumerated below.

- It is very hard to write any program which implements either a framework, such as Poligon or an application such as those which have been mounted on Poligon. This is due largely to asynchronous side effects. A system with better formal properties would be less error prone in this respect but might well make much less efficient use of the hardware. These difficulties could also be caused by an insufficiency of mechanisms to control coherency in Poligon.
- In order to produce a reliable program it is necessary to write code which makes no assumptions about anything that any other part of the system might be doing. Failure to do so results in brittle systems.
- In order to achieve a coherent solution it was found to be necessary to develop a number of programming methodologies.

*Node Level*      The creation of Nodes is tricky. Because each element is likely to represent some real-world object, such as an aircraft, it is important either to provide a mechanism for resolving the conflict caused by multiple asynchronous requests to create an element that represents the same thing or to provide a mechanism for managing the creation of Nodes. Poligon opts for the latter approach.

*Slot Level*      The programmer should cause each Node to have an idea of how to improve its own idea of the solution - to have *Goals*. In Poligon this is done at a fine grain, with each field of each element in the solution being able to have associated with it functions which enable it to evaluate itself.

It was found that a good axiom for programming these systems is "Never throw away any data unless you are convinced that you have better data." This is the sort of behavior that is used in the evaluation functions mentioned above.

*Rule Execution*      Poligon attempts to maintain the smallest critical sections possible. The original implementation of Poligon in fact had as its only atomic actions reading a field and writing a field. It was soon found that, in order to maintain consistency during rule execution, it had to be possible to read the values from a number of fields simultaneously - taking a snapshot without the subject moving. This, coupled with critical sections for the writing of collections of values, allows confidence that the picture that one sees when taking such a snapshot of a Node is consistent, even if not necessarily the most up to date. It is important for a Poligon programmer to be aware that the Node of which a snapshot has been taken may well be read from and written to by other Rules asynchronously during the invocation of the Rule taking the snapshot.

## 2.4 CAGE Problem Solving Framework

CAGE [KSL 86-31, KSL 88-02] is a framework for building and executing applications as a concurrent blackboard system. CAGE is based on the AGE (KSL 80-29) serial blackboard framework. It includes mechanisms for the concurrent execution of knowledge sources, rules and parts of rules. The CAGE user has complete control over which of these mechanisms are used. CAGE is designed to execute on a shared-memory, multiprocessor system with tens to

hundreds of processors. It is implemented using Qlisp, a concurrent dialect of Lisp designed for multiprocessors with a single, shared address space. CAGE currently executes on a shared-memory variant of CARE (see Section ) simulated using the SIMPLE simulation system.

#### 2.4.1 CAGE Design

CAGE is a blackboard framework system. In addition to the basic functionality found in AGE, CAGE allows user-directed control over the concurrent execution of many of its constructs. Otherwise, the two systems are functionally identical. The basic components of a system built with CAGE are:

- A global data store (the blackboard) on which emerging solutions are posted. The elements on the blackboard are organized into levels and represented as a set of attribute-value pairs.
- Globally accessible lists on which control information is posted (e.g. lists of events, expectations, etc.).
- An indefinite number of knowledge sources, each consisting of an indefinite number of condition-action rules.
- Various kinds of control information that determine (a) which blackboard element is to be the focus of attention and (b) which knowledge source is to be used at any given point in the problem solving process.
- Declarations that specify the components (knowledge sources, rules, condition and action parts of rules) to be executed in parallel, and when to force synchronization.

Using the concurrency control specifications, the user can alter the simple, serial control loop of CAGE by introducing concurrent actions. CAGE allows parallelism ranging from concurrently executing knowledge sources all the way down to concurrent actions on the condition and action sides of the rules.

#### 2.4.2 Building applications in CAGE

The CAGE System provides a CAGE language with which the user can write an application. The type of user-supplied information is similar to that required for applications constructed in the AGE system, however, the structure of the information is somewhat different.

##### Blackboard Data Structure

There are two major components in the CAGE blackboard structure, the hypothesis *classes* (frequently called levels in hierarchical blackboard structures) and the hypothesis *nodes*. The user must specify the classes that make up his application's blackboard structure. For each class, the user must define the fields to be associated with the nodes created in that class. Nodes are created in those classes, either a priori by the user or dynamically while executing the user's rules. Each of the classes is defined as an object with the attributes as instance variables and with the nodes as instances of the class objects.

##### Control Structure

All CAGE control information is referenced through the Control-Structure object which is basically the same as in AGE.

##### Knowledge Sources

CAGE knowledge sources are partitions of the application knowledge. Each knowledge source consists of some declarative information and a set of rules.

**Knowledge Source Declarations** A knowledge source consists of more than just groups of rules. In order to interpret the rules properly, CAGE needs answers to some questions about knowledge source control, for example,

- Under what circumstances should this knowledge source be invoked?

- Which one, of all of the rules whose condition part is satisfied, should be executed?
- Are there any local variables to be defined for this knowledge source?

The following are the primary knowledge source control options available for the user to use in order to tailor a knowledge source:

**Preconditions:** A list of tokens, representing the *event names* used in rules. If the currently focused event has an event name that matches one of the knowledge source's preconditions, then that knowledge source is activated.

**Hit Strategy:** There are two main hit strategies available in CAGE, Single and Multiple. When a knowledge source with a single-hit strategy is invoked, the rules of that knowledge source are evaluated, in order, until one rule's condition is satisfied. Then, the actions of the action part of the rule are executed, and no further rule is evaluated. With a multiple-hit strategy, the condition parts of all the rules are evaluated, and all the action parts of the rules whose conditions were true are executed.

**Definitions:** A list of local variables. The definitions are an efficiency feature to avoid the repeated calculation of the same variable. The structure is similar to that of LET, pairs of a variable names and expressions.

**Rule Order:** A list of rule names, representing the rules of the knowledge source. This is the order in which the rules are to be evaluated when in serial mode.

### Rules

CAGE rules consist of three major parts: definitions, conditions, and actions.

**Definitions:** The definition part of a rule is similar to a LET in structure. The scope of the variables defined here is the rule, both in the condition and action parts, as well as other definitions in the rule.

**Condition part:** The condition part consists of one or more conditional clauses. The clauses can be an arbitrary expression. The condition part can reference both the variables local to the rule or to the knowledge source. The CAGE system provides several access functions for retrieving values from the blackboard nodes which can be used in the condition part.

**Action part:** The action clauses make up the final part of a CAGE rule. The actions specify the changes to be made to the blackboard and how those changes are to be made. The user must specify what node and attributes on the blackboard are to be changed, what the new links or values are, and how those changes are to be made (possibly deleting some old values). The user must also specify an event name representing the type of change this action makes to the blackboard. If and when the event created by this action is selected as a focus event, this token will be matched against the preconditions of the knowledge sources to determine which knowledge source to invoke next.

### 2.4.3 Specifying Concurrency

CAGE supports the concurrent evaluation of various pieces of knowledge. The use of knowledge sources to partition the knowledge in blackboard systems and, in particular, the structure of the knowledge sources in CAGE provide several obvious places for concurrency. The knowledge sources group the domain knowledge into independent modules, which, theoretically, could be invoked independently and concurrently. Within each knowledge source the rules provide another source of parallelism, and within each rule, the clauses of the condition part and the different actions within the action part provide others. Of course, not all the clauses, rules or even knowledge sources are actually implemented totally independently of each other and some serialization may be necessary to solve the application problem correctly.

The following are the options for parallelism available in CAGE, grouped according to their allowed use in combination.

**Clause level:** can be used in combination with each other or any other parallel option.

**actions:** Execute the action clauses of a rule in parallel. Note: When running the actions concurrently non-determinism may result if both destructive (Supersede in CAGE) and constructive (Modify) actions occur to the same object-attribute.

**conditions:** Evaluate the condition clauses of a rule in parallel. Note: Use the rule definitions to set any local variables tested here, insuring that the lhs clauses will not be contending for the same data element.

**rule-definitions:** Evaluate the definitions of a rule in parallel. Again, these definitions should be independent of each other AND should avoid accessing the same data, if their concurrent evaluation is to result in an actual speed-up.

**Rule level:** Definitions can be used in combination with any of the other options, but only one of the rule options, single, multiple, sync or nosync can be used at a time.

**definitions:** Evaluate the definitions concurrently at the beginning of a knowledge source.

**rules-single:** Evaluate all the condition parts of the rules of a knowledge source concurrently, but only execute the actions of one successfully evaluated rule.

**rules-multiple:** Evaluate all of the conditions of the rules of a knowledge source concurrently, wait until all the evaluation is completed, then execute the actions of all the successfully evaluated rules serially.

**rules-sync:** Evaluate all the condition parts of the rules of a knowledge source concurrently, wait until all the evaluation is completed, then execute the actions of all applicable rules concurrently.

**rules-nosync:** Evaluate the condition parts of the rules of a knowledge source in parallel and execute the action part of each rule as soon as the conditions evaluate to true. Executed the actions within the action part in parallel. With this option there is no synchronization between the rules in the knowledge source.

**Knowledge source level:** Only one of the concurrency options for the knowledge source can be set at any one time.

**kss:** Activate all the applicable knowledge sources at once. Synchronization is accomplished by waiting for all knowledge sources to complete execution (and the event list is updated) before invoking a new set of knowledge sources concurrently.

**kss-nosync:** Invoke all applicable knowledge sources as soon as a new event is created. This option provides the least control of all the options available and does no synchronization. Many applications will have to be significantly changed to execute correctly under these conditions, particularly removing any possible circular knowledge source invocations. Without any synchronization, as soon as an event is created all relevant knowledge sources become active -- no events are added to the eventlist and no focus event is ever selected.

**kss-minisync:** Add an event to the event list and do minimal computation at the point of synchronization before invoking the next set of knowledge sources. The main computation done is the collection and pruning of similar events, leaving fewer events to activate subsequent knowledge sources.

#### 2.4.4 CAGE Machine Model

Because CARE is a message passing, distributed memory model, we had to create a shared memory variant of CARE to simulate CAGE execution. Currently we simulate an even number of processors, using half as processor-cache pairs and half as controller-memory pairs. The atomic unit of memory access in CAGE is a blackboard node. Concurrent node access requests are handled by simple spin lock mechanisms.

With CAGE-CARE every step of the simulation, down to a very low level, is measured. For example, one can track the length of the memory queues to get a handle on a major issue in programming concurrent blackboard systems, memory contention. Other measurable factors include the overhead for creating new processes, network communication costs and the cost of creating a new node. Using CAGE-CARE one can experiment with multiprocessors of various sizes and can get a reasonably accurate picture of the parallelism obtainable for a particular application. The only disadvantage for the user is the length of real time it takes to run a simulation on CAGE-CARE. and combinations later.

### 2.5 CAGE, Poligon and LAMINA Comparative Experiments

During the past contract period we have been developing application software and machine architecture models to support a series of end-to-end experiments comparing various concurrent programming systems for knowledge-based applications. The goals of these experiments are to:

1. Obtain quantitative comparisons of the performance of the programming systems.
2. Gain insights into how different concurrent programming models lead to different (or similar) application decomposition and organization.
3. Force the refinement of the concurrent programming systems so as to better support application development.
4. Gain insights into the ease or difficulty of writing application code in each of the programming systems.

#### 2.5.1 The Experiments

The common application for these experiments is Elint (KSL 86-69), a real-time, knowledge-based system for integrating pre-processed, passively acquired radar emissions from aircraft. This Elint application has been implemented in three different concurrent programming systems:

- The concurrent object-oriented programming model supported by LAMINA (see Section ). LAMINA is the basic, low-level programming interface to CARE, a grid-based, distributed address space, message passing multiprocessor architecture (see Section ).
- The Poligon system (see Section ). Poligon is a demon-driven system derived from the blackboard model of problem solving.
- The CAGE system (see Section ). CAGE is a concurrent descendant of the AGE serial blackboard framework.

Each of the implemented applications will be executed and evaluated using various input data sets and varying numbers of processors.

Application code written in either LAMINA or Poligon compiles to code which executes on the CARE architecture. CAGE, however, is targeted toward a single address space, shared variable multiprocessor architecture. CAGE is implemented in QLisp, a concurrent Lisp for shared variable multiprocessors. To support CAGE we had to develop a multiprocessor "blackboard machine" variant of CARE. This blackboard machine models a shared variable

architecture and includes the mechanisms and instruments necessary to manage and study memory contention. The architecture implements the blackboard and the control data structures in global, shared memory. It directly supports the CAGE system and application code written in QLisp.

### 2.5.2 Experiment Status

During the past contract period we have:

1. Completed the implementation of the the Elint application in each of the three concurrent programming systems.
2. Completed the development of the blackboard machine variant of CARE.
3. Developed an experiment plan for the comparative studies.
4. Developed a new measure of speedup as a function of the number of processors in a multiprocessor system. This measure is useful for evaluating system performance of real time applications and is based on the concept of maximum sustainable input data rate.
5. Completed the first set of experiments for each of the three programming systems.

## 2.6 The AIRTRAC Application

AIRTRAC (KSL 86-20) is the primary application driving our development of concurrent knowledge-based system programming methodologies. Also, it is one of the basic applications used for our multiprocessor architecture performance experiments. AIRTRAC is a knowledge-based signal interpretation and information fusion system. The system attempts to identify, track, and predict the future behavior of aircraft. In particular, it attempts to recognize aircraft which might be engaged in covert activity, for example, smuggling. The inputs to AIRTRAC are periodic radar tracking system reports, a priori, filed flight plans for some aircraft, and occasional intelligence reports about suspected covert activity.

AIRTRAC is designed to be sufficiently complex and realistic to adequately test various ideas about concurrent problem solving on multiprocessor machine architectures. The AIRTRAC application involves continuous input data streams, typical of real-time signal interpretation problems. Such problems often require a level of computational power two to three orders of magnitude beyond what is currently available. Moreover, the application uses data-driven, expectation-driven and model-driven styles of reasoning. These reasoning styles encompass a wide range of paradigms in artificial intelligence.

### 2.6.1 Overall Application System Structure

The overall system consists of radar collection sites and associated trackers, filed flight plan sources, intelligence report sources, and the AIRTRAC system running on a multiprocessor.

Output from each radar is fed to an associated tracker which produces periodic track reports for input to AIRTRAC. A tracker detects aircraft, estimates their positions and velocities, and assigns unique track identifiers. A tracker continues to assign the same identifier if it believes that the received signal is due to the same aircraft which was previously seen. Periodic reports from the tracker include the scantime, track identifier, and the mean and covariance of the position and velocity of the track. Because of tracker limitations, they usually lose a track when the corresponding aircraft makes a significant maneuver such as turning sharply. A tracker assigns different identifiers to the tracks before and after such a maneuver. One of the tasks of AIRTRAC is to connect such "broken" tracks. Another AIRTRAC task is to fuse multiple tracks which represent the same aircraft observed from different radar sites.

A filed flight plan is information regarding the expected position at given times of the flight path of an aircraft. Since filed flight plans are only estimates of actual flight paths, their

track information is less precise than actual observed track data. Filed flight plans are usually available for cooperative aircraft. Intelligence reports provide information about possible origins, possible destinations, and possible flight times for aircraft engaged in covert activity. This information typically embodies a "tip-off" about covert activity. Due to the sketchy nature of the information, intelligence reports are even less precise than filed flight plans. AIRTRAC attempts to fuse observed tracks, filed flight plans, and intelligence reports which represent the flight path of the same aircraft.

### 2.6.2 AIRTRAC Organization

The AIRTRAC system is partitioned into three major modules. At the lowest level of data abstraction, the Data Association Module accepts as input the periodic output of the radar trackers. The primary task of the module is to abstract the periodic track reports into sequences of straight-line Radar Track Segments that represent (approximately) constant-heading, constant-velocity segments of an aircraft's flight path. Other tasks of this module are to recognize when a track with a new identifier is initiated, determine when sufficient evidence has been collected for a track to confirm its existence with a given probability, and to recognize when a track with a given identifier has been terminated.

The Path Association Module receives the Radar Track Segments from the Data Association Module. It attempts to "connect" the segments into coherent tracks representing the flight paths of the aircraft under observation. It then attempts to fuse the tracks which correspond to the same aircraft observed from different radar sites. The module also accepts as input filed flight plans and intelligence reports, and it attempts to fuse the plans and reports with the observed tracks. The module uses models of aircraft performance characteristics such as velocity, acceleration and maneuverability to help form hypothesized flight paths. The Path Association Module must deal with ambiguous data, and it maintains, if necessary, alternative flight paths for an observed aircraft. For each alternative, hypothesized flight path, the module maintains a measure of confidence in the hypothesis which rises as more evidence is accumulated fitting the hypothesis and which falls if expected behavior consistent with the hypothesis does not materialize.

The primary tasks of the Path Interpretation Module are to predict the future behavior of observed aircraft and to identify aircraft which are engaged or might engage in covert activity. The module takes into account the current and predicted flight paths of the observed aircraft, information about existing airports, known radar shadow regions, known flight corridors, and geographic and/or political boundaries. It uses models of aircraft behavior that embody strategies and goals to help form reasonable hypotheses.

### 2.6.3 AIRTRAC Status

The AIRTRAC Data Association Module and associated experiments were completed during past contract periods (KSL 87-61). The experiments were performed using the SIMPLE/CARE multiprocessor simulation system. They demonstrated that almost linear speedup as a function of the number of processors can be achieved (at least up to 100 processors) for a periodic data-driven knowledge-based system such as the Data Association Module.

During the past contract period, the design and knowledge acquisition for the Path Association Module was completed. Over one half of the LAMINA code for this module has been implemented and debugged.

## 2.7 Multiprocessor Load Balancing Studies

One of the more difficult problems in actually realizing high levels of concurrent execution of applications on multiprocessor systems is that of processor and/or memory load balancing. Based on our experiments with concurrent knowledge-based systems, the single largest impediment to achieving high utilization of multiprocessing resources is localized processor and/or memory "hot spots." That is, processors or memory access queues which are overloaded

relative to the rest of the system. Such hot spots result in many of the processors sitting idle awaiting information from the overloaded resources. This load balancing problem is particularly acute for concurrent applications such as signal interpretation where there is significant dynamic (i.e., run-time) creation and destruction of processes and data structures. This situation is in contrast to well-structured applications such as finite element computations where all processes and data structures are known at load-time.

### 2.7.1 Load Balancing Studies Status

Our work to date on load balancing has focused on non-adaptive schemes. That is, schemes in which once a process is allocated to a processing site it remains there throughout its life. In adaptive schemes active processes can migrate between processing sites.

For our earliest ELINT-CAOS experiments (KSL 86-69), we used an extremely simple load distribution scheme based on round-robin assignment of dynamically created objects to processing sites. This scheme resulted in poor resource utilization, for example, at best 25% average processor utilization for a 49 processing site CARE architecture.

We next experimented with various dynamic load distribution schemes employing techniques such as each site keeping track of its (logically) immediate neighbor's loads and using application domain knowledge to predict the lifetime and busyness of dynamically created objects. These schemes resulted in, at best, very marginal improvement over the round-robin scheme.

We then experimented with non-adaptive schemes based on random scattering of dynamically created objects to processing sites. Surprisingly, this scheme performed remarkably well relative to the earlier, more information intensive schemes. We are currently using a variant of the random scattering scheme in which each processing site is assigned an a priori preference weight with respect to accepting dynamically created objects. These weights are based on the distribution of load-time created objects onto sites. The random distribution of dynamically created objects to sites is skewed so as to respect this weighting.

Although this weighted random distribution scheme provides the most balanced loads that we have achieved to date, it still results in significant underutilization of machine resources. For example, we have achieved, at best, only about 50% average processor utilization on 64 site CARE architectures.

## 3 Spatial Reasoning

During the contract period we initiated development of an integrated system for possible export of PROTEAN. The ultimate goal of this portion of the work is a software package that provides a flexible framework for representing and using knowledge about protein chemistry and protein structures. It will include methods for general geometric constraint satisfaction and facilitate their use for protein structure determination. We expect to have most pieces of this package available for computer systems running the UNIX(TM) operating system by early 1989.

A paper is being written summarizing PROTEAN from an AI point of view, where the general issue is the problem of scaling up large-scale constraint satisfaction problems. The conclusions of that paper (and of the current phase of PROTEAN development) are that no single representation or algorithm is adequate, that simplifying assumptions and abstract models will be required, and that the role of heuristic reasoning is to choose the most appropriate representations, algorithms and control strategies during the problem solving.

## 4 Reasoning Under Uncertainty

This research focuses on methods for reasoning with uncertainty that are consistent with the Bayesian or decision theoretic paradigm. Although most researchers appreciate the benefits of

a normative approach, many have turned away from the Bayesian paradigm because they believe it is impractical to use in the construction of *large* expert systems. Over the last year, David Heckerman has been developing a representation language consistent with the axioms of decision theory that facilitates the construction of such large reasoning systems.

A common thread in much of the research in probabilistic reasoning over the past two decades is that the representation of conditional independence can be used to simplify knowledge acquisition and computation. The central notion in our current work is that there are many forms of independence, other than conditional independence, that can be exploited to simplify reasoning.

Over the last year, we have identified various forms of independence and have designed a representation language that can accommodate these forms in a unified framework consistent with the axioms of decision theory. A discussion of the various forms of independence and how they can be represented is given in (KSL 88-07).

## 5 List of Publications

The reports listed below were written during the period of the contract, or are referenced in the previous discussion. Although some of the reports listed were not directly supported by contract funds, their publication was made possible in part by the work that was directly supported, e.g. KSL 87-47 reports on an expert system for construction site space planning, which uses concepts developed for the PROTEAN system and subsequently incorporated in the ACCORD framework for spatial arrangement and assembly tasks.

### 5.1 Concurrent Computer Architectures

#### HPP 80-29

H. Penny Nii; *An Introduction to Knowledge Engineering, Blackboard Model and AGE*, March 1980, 45 pages

#### KSL 86-19

J.P. Rice; *Poligon, A System for Parallel Problem Solving*, April 1986. To appear in: *DARPA proceedings, Asilomar 1986*. 16 pages

#### KSL 86-20

J.R. Delaney; *Multi-System Report Integration Using Blackboards*, March 1986. Submitted for publication to: *1986 American Control Conference*. 12 pages

#### KSL 86-31

Nelleke Aiello; *User-Directed Control of Parallelism; The CAGE System*, April 1986. To appear in: *DARPA Proceedings of April 1986*. 12 pages

#### KSL 86-36

STAN-CS-87-1148. Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura and Greg Byrd; *An Instrumented Architectural Simulation System*, January 1987.

#### KSL 86-67

Bruce A. Delagi, Nakul P. Saraiya, and Gregory T. Byrd; *LAMINA: CARE APPLICATIONS INTERFACE*, November 1987.

#### KSL 86-69

STAN-CS-86-1136. Harold Brown, Eric Schoen, and Bruce Delagi; *An Experiment in Knowledge-Base Signal Understanding Using Parallel Architectures*, October 1986. In: *Parallel Computation and Computers for AI*, J.S. Kowalik, Editor, Kluwer Publishers.

#### KSL 87-02

STAN-CS-87-1146. Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi; *A Point-to-Point Multicast Communications Protocol*, January 1987.

#### KSL 87-07

STAN-CS-87-1144. Gregory T. Byrd and Bruce A. Delagi; *Considerations for Multiprocessor Topologies*, January 1987.

**KSL 87-43**

STAN-CS-87-1166. Hiroshi G. Okuno and Anoop Gupta; **Parallel Execution of OPS5 in QLISP**, June 1987. Shorter version appeared in: *Proceedings of the Fourth Conference on Artificial Intelligence Applications (CAIA-88)*, IEEE, March 1988.

**KSL 87-44**

STAN-CS-87-1178. Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi; **A Dynamic, Cut-Through Communications Protocol with Multicast**, August 1987.

**KSL 87-61**

STAN-CS-87-1188. Russell Nakano and Masafumi Minami; **Experiments with a Knowledge-Based System on a Multiprocessor**, October 1987.

**KSL 87-65**

STAN-CS-87-1189. Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd; **Instrumented Architectural Simulation**, November 1987.

**KSL 88-02**

H. Penny Nii, Nelleke Aiello, and James Rice; **Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon**, in *Blackboard Systems*, Robert Englemore and Anthony Morgan, Editors, Addison-Wesley Ltd., Wokingham, England, 1988.

**KSL 88-04**

J. P. Rice; **Problems with Problem-Solving in Parallel: The Poligon System**, January 1988.

**KSL 88-10**

Gregory T. Byrd and Bruce A. Delagi; **A Performance Comparison of Shared Variables vs. Message Passing**, February 1988. In *Proceedings: Third International Conference on Supercomputing*, May 1988.

**KSL 88-25**

(Working Paper) Greg Byrd; **Modelling a Bus-Based Multiprocessor Using the CARE Simulation System**, March 1988.

**KSL 88-33**

Bruce A. Delagi and Nakul P. Saraiya; **ELINT in LAMINA: Application of a Concurrent Object Language (Extended Abstract)**, July 1988.

**KSL 88-41**

(Working Paper) Alan C. Noble and Everett C. Rogers; **AIRTRAC Path Association: Development of a Knowledge-Based System for a Multiprocessor**, June 1988.

**5.2 Spatial Reasoning****KSL 87-05**

STAN-CS-87-1142. James F. Brinkley, Bruce G. Buchanan, Russ B. Altman, Bruce S. Duncan, Craig W. Cornelius; **A Heuristic Refinement Method for Spatial Constraint Satisfaction Problems**, January 1987.

**KSL 87-47**

Iris D. Tommelein, Raymond E. Levitt, Barbara Hayes-Roth; **Using Expert Systems for the Layout of Temporary Facilities on Construction Sites**, October 1987. In *Managing Construction Worldwide, Vol. 1, Systems for Managing Construction*, Lansley, P.R., Harlow, P.A. (eds), pp. 566-577, E.&F.N. Spon, London, 1987.

**KSL 88-19**

Barbara Hayes-Roth, Micheal Hewett, M. Vaughan Johnson, and Alan Garvey; **ACCORD: A Framework for a Class of Design Tasks**, March 1988.

Barbara Hayes-Roth, Bruce Buchanan, Olivier Lichtarge, Micheal Hewett, Russell Altman, James Brinkley, Craig Cornelius, Bruce Duncan and Oleg Jardetzky; **PROTEAN: Deriving Protein Structure from Constraints**, in *Blackboard Systems*, Robert Englemore and Anthony Morgan, Editors, Addison-Wesley Ltd., Wokingham, England, 1988.

### 5.3 Uncertain Reasoning

**KSL 87-45**

(Journal Memo) David Heckerman and Holly Jimison; **A Perspective on Confidence and Its Use in Focusing Attention during Knowledge Acquisition**, in: *Proceedings of the Third AAAI Workshop on Uncertainty and Probability in Artificial Intelligence*, July 1987.

**KSL 87-46**

(Journal Memo) Michael P. Wellman and in: David E. Heckerman; **The Role of Calculi in Uncertain Reasoning**, *Proceedings of the Third AAAI Workshop on Uncertainty and Probability in Artificial Intelligence*, July 1987.

**KSL 87-53**

(Journal Memo) David Heckerman and Eric J. Horvitz; **On the Expressiveness of Rule-Based Systems for Reasoning with Uncertainty**, August 1987. In: *Proceedings of AAAI, Vol. 1, July, 1987*, pp 121-126.

**KSL 88-07**

David Heckerman; **Formalizing Heuristic Methods for Reasoning with Uncertainty**, May 1987.

David Heckerman; **An Empirical Comparison of Three Inference Methods**, to appear in: *Proceedings of the Fourth AAAI Workshop on Uncertainty and Probability in Artificial Intelligence*, August 1988.